CHALMERS
UNIVERSITY OF TECHNOLOGY

Type theory as a framework for modelling and programming

Citation for the original published paper (version of record):
Ionescu, C., Jansson, P., Botta, N. (2018). Type theory as a framework for modelling and programming. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 11244(1): 119-133.
http://dx.doi.org/10.1007/978-3-030-03418-4_8

Type Theory as a Framework for Modelling and Programming

Cezar Ionescu[1][0000−0003−3908−2843], Patrik Jansson[2][0000−0003−3078−1437], and Nicola Botta[3][0000−0002−8923−2734]

[1] University of Oxford, Cezar.Ionescu@conted.ox.ac.uk
[2] Chalmers University of Technology, patrik.jansson@chalmers.se
[3] Potsdam Institute for Climate Impact Research, botta@pik-potsdam.de

Abstract. In the context provided by the proceedings of the UVMP track of ISoLA 2016, we propose Type Theory as a suitable framework for both modelling and programming. We show that it fits most of the requirements put forward on such frameworks by Broy et al. and discuss some of the objections that can be raised against it.

Keywords: Software Technology · Specification · Functional Programming · Dependent Types · Domain-Specific Languages

1 Introduction

The present paper was written as a contribution to the ISoLA 2018 track entitled "Towards a Unified View of Modeling and Programming". The basic question to be discussed there was that of the relation between "modelling"[4] and "programming". In one of its stronger forms, the question was formulated as

> what are the arguments for and against the statement:
> *modeling is programming*?

Such a question needs to be addressed in a certain context, one which would exclude certain possible meanings of "modelling", e.g., "To display (clothes) as a fashion model" (9th entry in the current Oxford Dictionary of English under "modelling"), and of "programming" (e.g., "To arrange by or according to a programme; to include or name in a programme; to draw up a scheme or itinerary of; to plan or schedule definitely", first entry under "programming" – the computer-oriented meaning only comes in fourth place). In order to obtain this context, we made a review of the definitions of modelling and programming used in the previous edition of this track, hosted at ISoLA 2016. The results of this review are presented in the next section. As a result of this review, we were led to propose Type Theory as a unified framework for modelling and programming. We present a brief description of Type Theory, and show that the proposal is consistent with most of the requirements for such an framework put forward during ISoLA 2016 by Broy et al. We then discuss some possible objections, followed by some of the wider implications of this proposal.

[4] We have used the British spelling throughout the document, except in literal quotes.

2 ISoLA 2016 definitions of modelling and programming

Seven of the sixteen contributions published within the "Towards a Unified View of Modeling and Programming" section of the ISoLA 2016 proceedings [38] contain a more or less explicit definition for both "modelling" and "programming".

- Selić [51]:
  - An *engineering model* is a selective representation of some system intended to capture accurately and concisely all of its essential properties of interest for a given set of concerns.
  - A program is a human-readable textual representation of the binary data that is actually stored and executed in a computer.

  Remark: since a program is itself a "selective representation", it follows that it is also a model (cf. the given definition), but Selić argues that this is misleading, since "programming languages are intended primarily and almost exclusively for prescriptive purposes".

- Seidewitz [50]:
  - A model is always about something, which I term the system under study (SUS). For our purposes here, we can consider a model to consist of a set of statements about the SUS expressed in some modeling language. These statements make assertions about certain properties of the SUS, but say nothing about other properties that are not mentioned.
  - Programs [...] are precise models of execution (where, for simplicity, I consider both data and algorithmic aspects to be included in the term "execution").

  Remark: Seidewitz considers that models are more general than programs: "From this point of view all programs are actually models. And all executable models are actually programs. But there are, of course, software models that are not programs."

- Elaasar and Badreddin [15]:
  - A model is a simplified representation of a more complex system. It is frequently used to abstract and analyze a system by focusing on one or more aspects. Models are used to understand, communicate, simulate, calibrate, evaluate, test, validate and explore alternatives for system development. Modelers use a wide variety of models to explore different aspects of the system such as requirements, structure, behavior, event, time, security, flow, process, activity, performance, quality, usability, etc. These models can be expressed in many forms including textual and visual representations.
  - Programming, on the other hand, is the activity of developing executable software. Programs are written in a programming language, which is a set of rules for expressing computations in a human-readable form that can be translated unambiguously to a machine-readable form.

- Prinz et al. [46]:
  - Modelling is the activity to describe a real or imagined (part of a) system using a language with a semantics. The model does not provide a full match of the real system, but an abstraction.
  - Programming is the activity to prescribe a new (part of a) system using a language with a well-defined execution semantics. The program determines the system.

- Lethbridge et al. [36]:
  - Three criteria for what it means to "look and feel like a model", attributed by the authors to Ludewig [37, p. 196] and summarised as
    - (m1) There is a mapping between the model and the system being modeled, or part of it. The system is called the 'original' by Ludewig.
    - (m2) This mapping abstracts some properties of the system, hence providing a simplified view. Typical abstractions focus on behavioural properties or structural properties, but the same model may include both, as well as other types of abstractions.
    - (m3) The model is useful in that one can do things with the model instead of having to have access to the full (executable) system. Key things one can do with a model under m3 include analyzing it to measure it or to find defects, and transforming it into other forms. Models are therefore useful in early stages of design, but in some cases can also be used to generate some or all of the system.
  - Three criteria for what it means for a system to "look and feel like code":
    - (c1) The system, or parts of it, are composed of a set of units (files in the case of Umple), which can be edited using a text editor supporting syntax highlighting.
    - (c2) The textual syntax is designed to be usable by programmers.
    - (c3) When it is processed (compiled in the case of Umple), feedback such as warnings and errors is produced, highlighting issues on specific lines.

- Naujokat et al. [42]:

  > At a conceptual level, modeling and programming can be regarded as two sides of the same medal: the WHAT and the HOW descriptions of a certain artefact. This duality of WHAT and HOW has a long tradition in engineering, where models were built to predict certain WHATs, like the aerodynamics of an envisioned car or its visual appearance, in order to optimize vital aspects, before entering the costly HOW-driven production phase, where modifications become extremely expensive. [...] In classical engineering, there is typically a very clear and agreed upon distinction between a model (a WHAT) and an implementation (the HOW), frequently connected to distinct abstraction layers and different natures of the respective description means. [...] the understanding of what is a HOW (an implementation or a program) and what a WHAT (a model or a specification) in software becomes quite situation dependent.

– Broy et al. [12]:

> Models are meant to describe a system at a high level of abstraction for the purpose of human understanding and analysis. Programs, on the other hand, are meant for execution. However, programming languages are becoming increasingly higher-level, with convenient notation for concepts that in the past would only be reserved for formal specification languages.

Of the remaining nine papers, Berry [4] defines programming ("As an activity, programming is quite easy to define: one writes texts or graphics that are compiled into some machine language and executed by some computer."), but not modelling ("Modeling is not as clear-cut, because it deals with many more concepts and objects.").

The remaining eight papers contain no definitions of modelling or programming. Rybicki et al. [49], and Larsen et al. [31] use the terminology of the model-based engineering community (see, e.g., [40]), Rouquette [48] that of UML [43], Elmqvist et al. [16] that of Modelica [18], Haxthausen and Pelska [20] "model" both in the sense of modelling languages and in that of model theory. Lattmann et al. [33] discuss domain-specific modelling languages. Kugler [28] refers to "a combination of programming, modelling languages, specification formalisms and methodologies", but there are no details in his brief contribution. Finally, Leavens et al. [34] do not mention modelling at all, discussing instead *specifications*, i.e., partial descriptions of a software system against which the correctness of *implementations* is to be assessed. This last treatment of modelling might seem quite limited when compared to the others, but we believe that in this context it is, in fact, quite natural.

2.1 Models and specifications

The picture that emerges from a study of the definitions and of the most important references given in the ISoLA 2016 proceedings is, broadly speaking, the following: a model of a system is a partial description of that system. The particular form of the partial description depends on the means we have, on the system to be modelled, on what we plan to do with the description, etc. In general, the relationship between the description and the system is not fully formalised (and often not fully formalisable). There is a great variety of kinds of description (scale models, mathematical models, narratives, pictures, formal models, software models, ...). The systems being modelled do not necessarily exist "in reality". For example, architectural blueprints can be seen as partial descriptions of buildings that have yet to be built. In such cases, the relationship to the system can be quite formal[5], as blueprints are part of contracts and it must be decidable whether the buildings have been constructed correctly or not.

[5] Here, *formal* is in the OED's sense 5.a: "Done or made with the forms recognized as ensuring validity; explicit and definite, as opposed to what is matter of tacit understanding."

This variety of possible models can seem quite daunting, but there is one definite constraint in the context of ISoLA: the only models being considered are those that describe programs. In other words, the only models considered are *specifications*, which justifies the point of view implicitly adopted by Leavens et al. in [34].

This constraint is obvious in most of the articles above, but what about the models of real systems considered by Kugler, Printz et al., and others? In these cases, the models are meant to describe *simulations* of the real systems, but simulations are the result of the execution of programs. This is explained in Printz et al. when they discuss correctness: a model of a system (called the reference system) is correct if programs described by the model produce a simulation of that system.

In their contribution, Broy et al. call for "a single universal formalism for modeling and programming any form of system", but, again, the context makes clear that what is meant are software systems (object-oriented, functional, imperative, etc.). The unification being sought is at the level of the *framework* used, ideally, for both activities. Most of the ISoLA 2016 papers refer to such a framework, sometimes called *environment* (as in Elmqvist et al. [16]) or thought of as a high-level programming language (as in Seidewitz [50]).

3 A brief introduction to Type Theory

If the above analysis is correct, and we can therefore equate modelling with specification of software systems, then that is good news, for we do have a unified framework for modelling and programming, one that is mature (several decades old), with solid implementations (NuPRL, Coq, Agda, Idris, Lean), and impeccable mathematical credentials: *Type Theory*.

Type Theory, sometimes referred to as Dependently Typed Theory, is a pure functional programming language with a static type system. It is similar to Haskell, and stands in roughly the same relation to it as predicate logic to propositional logic. Type Theory was developed by the Swedish mathematician and philosopher Per Martin-Löf, who intended it to have the same foundational role for intuitionistic mathematics that set theory expressed in predicate logic had had for classical mathematics.

This is not the place for a presentation of Type Theory, especially since nowadays there are many very good ones available (for a particularly accessible one, see [1]). What we want to do here is to provide an intuition for why Type Theory is able to provide an environment for both specifications and implementations, and for the various "types as ..." analogies.

We start by recalling that set theory derives its foundational role in classical mathematics from its ability to represent properties in several different (equivalent) ways, within a first-order language. For example, given a property P over a set A, expressed as a formula in the first-order language of sets, we can view it as a

- set $P = \{\, a \mid P\ a \,\}$, $a \in P$ iff a has the property P
- Boolean-valued function P: $P\ a = \mathit{True}$ iff a has the property P
- set-valued function P: $P\ a$ not empty iff a has the property P

In the third representation we can think of $P\ a$ as the set of *witnesses* to a having the property P.

All these allow us to talk about the property within the theory: it becomes an element of the universe of discourse. In contrast, the formula expressing the property is not an element of the universe of discourse.

These are consequences of the axiom of comprehension, which, in particular, directly legitimises the view of properties as sets. Other axioms of set theory introduce new ways of building sets from existing ones, by means of taking the powerset, unions and intersections.

If we take types in programming languages to be the analogues of sets in set theory, we can see that the available means for their construction are more restricted. Like many other programming languages, Type Theory allows the construction of inductive types. For example:

$$\frac{}{Z : \mathit{Nat}} \qquad \frac{n : \mathit{Nat}}{S\ n : \mathit{Nat}}$$

and

$$\begin{array}{l}\textbf{data}\ \mathit{Nat} : \mathit{Type}\ \textbf{where}\\ \quad Z : \mathit{Nat}\\ \quad S : \mathit{Nat} \to \mathit{Nat}\end{array}$$

are two equivalent ways of expressing the rules for the construction of natural numbers, one in "natural deduction" style, the other in the style of Haskell, Agda, or Idris.

In most programming languages, we can usually represent properties as Boolean-valued predicates. For example:

$$\begin{array}{ll}\mathit{isEven} : \mathit{Nat} & \to \mathit{Bool}\\ \mathit{isEven}\ Z & = \mathit{True}\\ \mathit{isEven}\ (S\ Z) & = \mathit{False}\\ \mathit{isEven}\ (S\ (S\ n)) & = \mathit{isEven}\ n\end{array}$$

In most cases, however, we cannot represent the associated set (here, the set of even numbers) as a datatype or as a type-valued function. Therefore, if a function requires its argument to be even, then the best we can do is to guard the call of the function with a run-time check. This leads to expressing requirements or specifications as tests, as in test-driven development methods or design by contract.

In contrast, in Type Theory, we have the additional possibility of representing a property by a type-valued function (or type family), which corresponds to the set-valued version in set theory. For example

$$\frac{k : \mathit{Nat}}{\mathit{MkEven}\ k : \mathit{Even}\ (2 * k)}$$

and

$$\textbf{data}\ \mathit{Even} : \mathit{Nat} \to \mathit{Type}\ \textbf{where}$$
$$\quad \mathit{MkEven} : (k : \mathit{Nat}) \to \mathit{Even}\ (2 * k)$$

are equivalent ways of expressing the type-valued function version of *isEven*. For every natural number n, *Even* n is a type. If n is not even, then the type will be empty. Otherwise, the type will have one element, namely *MkEven* $(n\ /\ 2)$. Perhaps the best way to think of an element $e : \mathit{Even}\ n$ is that it represents *evidence* that n is even, by showing that n is made out of the doubling of a natural number.

If a function requires its argument to be even, we can now formulate this requirement at the level of its type, for instance

$$f : (n : \mathit{Nat}) \to \mathit{Even}\ n \to X$$

The function f is here in *curried* form, allowing partial application: $f\ n$ is a function of type $\mathit{Even}\ n \to X$, $f\ n\ e$ is a value of type X, assuming n and e have the appropriate types. This notation is standard in functional programming languages, but also, e.g., in VDM (see the VDM-10 manual ([32]), Section 3.2.8, page 29).

In order to call f with an argument n, we have to supply another argument of type *Even* n. We can only do that if n is *Even*, since otherwise *Even* n would be empty. This additional argument must be reducible to the form *MkEven* k, where $k = n\ /\ 2$, and this can be checked at compile time (or, rather, at "type-checking time"). This ensures that f will never give rise to a run-time error, a much stronger guarantee than we can enforce by means of tests.

The ability to define inductive datatypes and type families lends Type Theory a surprisingly strong expressive power, equal to that of classical higher-order logic. In particular, we can formulate all the notions in current mathematics. Note, however, that the only formulas we can prove are those of constructive mathematics: the logic of Type Theory is intuitionistic.

When it comes to specifications of programs, this is not a bug, but rather a feature. The requirements on a program can be expressed at the level of types, for example

$$f : (x : X) \to \mathit{Pre}\ x \to \Sigma\ (y : Y)\ (\mathit{Post}\ x\ y)$$

is the type of a function that takes as input elements of a type X having the property *Pre*, and delivers elements of a type Y which are in the relation *Post* with the input. An implementation of f that satisfies the type checker will fulfil this specification.

This approach to specification and implementation in Type Theory has been successfully used in e.g., producing a verified C compiler, CompCert [35]; developing database access libraries which statically guarantee that queries are consistent with the schema of the underlying database [44]; implementing secure distributed programming [52]; implementing resource-safe programs [41,11]; and many others.

As a modelling framework, modelling in Type Theory has the advantage of mathematical consistency over using UML or similar approaches. As such, it is closer to formal methods like VDM, but we find it easier to express high-level, domain-specific properties in Type Theory. For example, we can formulate types for "resource-safe operations", "privacy-ensuring protocols", but also for "avoidable states" [6] or even "measures of vulnerability to climate change" [22].

4 Type Theory as a framework for modelling and programming

In [12], Broy et al. put forward ten requirements for a unified framework for modelling and programming. We give a brief overview of how Type Theory fares with respect to them.

1. *Target domains:* can the formalism account for "modeling; programming of non-embedded systems, such as web applications, including scripting; and finally programming of embedded and cyber-physical systems"? Type Theory has been successfully used in all these domains, for example: modelling using dynamical systems [23,6], implementing typed web client applications [27], programming embedded systems [47].
2. *Predicate specifications:* "A formalism must generally support specifying properties as predicates rather than only as algorithms." As explained above, we can use types to express arbitrarily complex predicates.
3. *Programming in the large:* "A formalism must support programming in the large, and in general provide good modularization and component-based development." Types are a natural structuring mechanism for programs, especially when supplemented with higher-order constructs such as type classes. Most implementations of Type Theory provide support for modules, separate compilation units, packages, etc. In the context of "programming in the large", Broy et al. emphasise *concurrency*, both at the level of programming and that of modelling. Concurrent programming is difficult in any framework, and Type Theory is no exception, but it is the topic of active research (see, for example, [10,27,21]) that can build on the high-quality Haskell implementations of concurrency [39].
4. *High-level programming:* "A formalism must support high-level programming as found in modern programming languages." Type Theory *is* a high-level programming language, so this requirement could be considered satisfied "by definition". However, the explanation of this requirement notes: "A formalism should be statically typed, although with type inference, and with allowance for going type less in clearly defined regions to support scripting." While it is not obvious to us that scripting necessarily implies dynamic typing (especially in the presence of type inference), we do believe that there are situations in which the type checker must be forced to accept a given typing. This is the case, for example, when integrating with external programs written in a different language, or for which the source code is not available. Most

(all?) implementations of Type Theory provide such a mechanism, usually by means of *postulates*.

5. *Low-level programming:* "A formalism must support low-level programming." The dependently-typed language Low* has been used to implement efficient low-level programs [47], and there are many other similar applications of implementations based on Type Theory. However, it is correct that at the current stage, implementations of Type Theory do not generate programs with the same performance characteristics as C.
6. *Continuous mathematics:* "A formalism can support modeling of cyber-physical systems." As a system originally designed for the formalisation of mathematics, Type Theory fulfils this requirement "by construction". The modelling capabilities of constructive mathematics have been amply demonstrated, e.g., by Bishop and Bridges [5], and the ForMath project [17].
7. *Domain-specific languages:* "A formalism must support definition of domain-specific languages." Like most functional programming languages, Type Theory is an excellent vehicle for embedding domain-specific languages (see, for example, Brady [9]).
8. *Visualization:* "A formalism must be visualizable.". This requirement is the only one that is currently not satisfied. Providing visual representations of formal specifications such as those represented by types in Type Theory is a problem that not only has not been solved, but, as far as we know, is not currently being tackled in a systematic way (say, in the framework of a Horizon 2020 project). The activities that come nearest to the mark are those involving diagrammatic reasoning, such as string diagrams [14]. These offer visual representations of the relationships of various entities in a categorical setting, and come with rules that allow rigorous proofs by means of manipulations of the diagrams. There exists a software tool that implements this kind of reasoning with string diagram, available at http://globular.science/ [2], but it is unclear whether this kind of presentation would be appropriate for the proofs normally conducted in Type Theory. Perhaps the main difficulty here is that of coming up with the "right kind" of visualisation of type-based specifications, which will require the joint effort of HCI experts, modellers, programmers, and specialists in Type Theory.
9. *Analysis:* "A formalism must be analyzable.". This requirement refers to "basic built-in support for unit testing, over advanced testing capabilities, including test input generation and monitoring, to concepts such as static analysis, model checking, theorem proving and symbolic execution", which the more popular implementations of Type Theory support. However, the requirement asks that "the main emphasis should be put on automation. The average user should be able to benefit from automated verification, without having to do manual proofs." Tactics implemented in, e.g., Idris or Coq, attempt to automate certain parts of proofs, and are quite successful when dealing with properties that fit a certain pattern (which is often the case in DSLs, [11]). However, the moment one strays from the beaten path, proof obligations can no longer be filled-in automatically, thus we can only claim partial satisfiability of this requirement.

10. *What modelers do that programmers don't:* "A central question is how a model/program is represented." This requirement refers to the need for "a more sophisticated approach than the text-based source code repositories often used by programmers", since modelers "have the habits of querying models, transforming models, and generally consider models as data, in contrast to the programming community where data usually are separated from programs", and notes that "from within a program one can usually not get access to the entire AST of the program itself, although often limited forms of reflection are possible". Frameworks based on Type Theory are among the leading environments for *meta-programming* (a term that covers both reflection and code generation), which has been considered one of the "killer applications" for dependent types (e.g., by Chlipala in [13]), so we consider that this requirement is satisfied. Broy et al. point out that this requirement is connected to that of visualisation: this link might provide a starting point for projects aiming to satisfy the latter.

Type Theory fully satisfies most of the requirements, with partial scores for "programming in the large", "analysis", and "what modelers do". The only requirement that is not satisfied is "visualisation", which we hope will be a topic of future research.

5 Potential objections

In this section, we consider some potential objections to using Type Theory as a unified framework for modelling and programming.

Three of the papers of the ISoLA 2016 proceeding, Haxthausen and Peleska [20], Larsen et al. [31], and Naujokat et al. [42], argue against the feasibility and usability of a unified framework for modelling and programming. In all three, the argument is that multiple formalism are needed to do justice to the wealth of potential goals, requirements, stakeholders etc.

All three papers point out the necessity of relating the various formalisms, in order to combine them to create more complex models, or to translate between them in order to reuse common aspects. We believe that the best way to do this is to implement the various formalisms as DSLs embedded in a common language, and that Type Theory is the most adequate candidate for such a language.

We will, however, very quickly admit that it is not a *perfect* candidate. The required level of precision can sometimes become a burden. For example, since each value has a unique type, we have difficulties working with subtypes. This can create problems when building *hierarchies* of models, since the familiar "subset" relation turns out to be quite awkward in a type-theoretical context. Similar remarks apply to other common set-theoretical constructions, such as that of *quotient sets*, which amount to introducing a new equality relation on a set. In Type Theory, the canonical equality on the elements of a type, namely the identity relation, has a privileged status, and working with a different equivalence relation instead is much more cumbersome.

Type Theory is an area of active research, and we hope that these difficulties will gradually be alleviated. In particular, the developments in the area of *homotopy type theory* seem to hold the key to the problem of working with different equivalence relations.

6 Conclusions

We have presented several arguments for the use of Type Theory as a framework for unified modelling and programming, where we have interpreted "modelling" to refer to (partial) descriptions of software systems, i.e., specifications of software systems.

In these concluding remarks, we would like to explain why we believe that Type Theory is a more adequate such framework than others, such as VDM or the B-method. It is quite likely that, with some additions and modifications, these too could meet the requirements put forward by Broy et al. After all, VDM and similar frameworks have been developed for the exact purpose of covering the spectrum from software specification to implementation in a formal, systematic fashion. Moreover, they have the same mathematical foundation as all (or at least most) of classical mathematics: set theory.

Indeed, all the standard mathematical theories can be "compiled down" to the first-order language of ZFC (Zermelo-Fraenkel with the Axiom of Choice). However, ZFC is far from actual mathematical practice. Instead, what one usually sees is a usage of "naive" set theory, as presented in the books of Halmos [19] and Bourbaki's summary [7] (but *not* in Bourbaki's extended treatment of set theory [8]!). This is then combined with some form of "naive" (and mostly implicit) type theory, to prevent set-theoretical "excesses", such as taking the intersection of π with the square root function. This has been pointed out again and again, and has led to the search for alternative foundations, e.g., based on category theory. For the computer scientist, this comes as no surprise: after all, just because every programming language must eventually be compiled down to machine code, it does not at all follow that the best way to understand programming languages is through the prism of machine code.

Thus, perhaps surprisingly, being based on ZFC offers little advantage when it comes to modelling actual mathematical concepts (see the requirement labelled *continuous mathematics* in Broy et al.'s list). On the contrary, the awkwardness in formulating and working with notions such as "continuous function", "differentiable function", "linear operator", etc., makes it difficult for such systems to make inroads into the area of scientific computing.

ZFC is also at a disadvantage when it comes to the foundations of computing science, for example, in giving an account of the semantics of programming languages. In fact, the study of the relationships between various programming languages has led to the introduction in computing science of the lambda calculus [30] and its various typed variants [3,45]. This has influenced the current style in computing science, which emphasises the distinction of syntax versus semantics,

the introduction of names and structure [29], encourages calculational proofs and the creation of DSLs, all using types as the main structuring mechanism.

The mathematician Charles Wells used the term "computer science perspective" in an article published in the American Mathematical Monthly [54], in which he was arguing that this style could also be valuable in teaching mathematics. This "perspective" is perhaps one of the most valuable contributions that computing science can make to the larger intellectual landscape, and we have witnessed its effectiveness during the lectures given within the *Domain-Specific Language of Mathematics* course taught in Chalmers from 2015/16 on [24,25,26].

Type Theory provides a natural foundation for both the computer science perspective and for constructive mathematics. When extended with classical postulates, resulting in a *typed predicate logic* [53], it brings us much closer to the language of mathematical practice than ZFC. Thus, Type Theory turns out to be a suitable vehicle for both mathematics and computing, at least in part because it was *not* created with any connection to software development.

A final remark: we have consciously decided to talk about "Type Theory" rather than any one of its implementations, because the most important unification that can be achieved is at the *conceptual* level, rather than the software level. The existing implementations have their strengths and weaknesses, and readers should make their choice based on their goals, needs, and background.

Acknowledgements

The work presented in this paper heavily relies on free software, among others on Idris, Agda, GHC, git, vi, Emacs, LaTeX and on the FreeBSD and Debian GNU/Linux operating systems. It is our pleasure to thank all developers of these excellent products. This work was partially supported by the CoeGSS project (grant agreement No 676547), which has received funding from the European Union's Horizon 2020 research and innovation programme.

References

1. Altenkirch, T.: Naive type theory (2017), http://www.cs.nott.ac.uk/~psztxa/mgs-17/notes-mgs17.pdf, lecture notes for a course at MGS 2017.
2. Bar, K., Kissinger, A., Vicary, J.: Globular: an online proof assistant for higher-dimensional rewriting. Logical Methods in Computer Science **14**(1) (2018), doi: 10.23638/LMCS-14(1:8)2018, http://arxiv.org/abs/1612.01093
3. Barendregt, H.P.: Lambda calculi with types. In: Abramsky, S., Gabbay, D.M., Maibaum, S.E. (eds.) Handbook of Logic in Computer Science (Vol. 2), pp. 117–309. Oxford University Press, Inc., New York, NY, USA (1992), http://dl.acm.org/citation.cfm?id=162552.162561
4. Berry, G.: Formally unifying modeling and design for embedded systems - A personal view. In: Margaria and Steffen [38], pp. 134–149, doi: 10.1007/978-3-319-47169-3_11
5. Bishop, E., Bridges, D.: Constructive Analysis. Springer-Verlag (1985), doi: 10.1007/978-3-642-61667-9

6. Botta, N., Jansson, P., Ionescu, C.: Contributions to a computational theory of policy advice and avoidability. Journal of Functional Programming **27**, 1–52 (2017), doi: 10.1017/S0956796817000156
7. Bourbaki, N.: Éléments de mathématique: Fasc. I. Livre 1, Théorie des ensembles;[5], Fascicule de résultats. Hermann (1964)
8. Bourbaki, N.: Théorie des ensembles. Springer (2006)
9. Brady, E.: The idris programming language — implementing embedded domain specific languages with dependent types. In: Central European Functional Programming School - 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers. pp. 115–186 (2013), doi: 10.1007/978-3-319-15940-9_4
10. Brady, E.: Type-driven development of concurrent communicating systems. Computer Science **18**(3) (2017), doi: 10.7494/csci.2017.18.3.1413, https://journals.agh.edu.pl/csci/article/view/1413
11. Brady, E., Hammond, K.: Resource-safe systems programming with embedded domain specific languages. In: Practical Aspects of Declarative Languages, pp. 242–257. Springer (2012), doi: 10.1007/978-3-642-27694-1_18
12. Broy, M., Havelund, K., Kumar, R.: Towards a unified view of modeling and programming. In: Margaria and Steffen [38], pp. 238–257, doi: 10.1007/978-3-319-47169-3_17
13. Chlipala, A.: Ur: Statically-typed metaprogramming with type-level record computation. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 122–133. PLDI '10, ACM, New York, NY, USA (2010), doi: 10.1145/1806596.1806612
14. Coecke, B., Kissinger, A.: Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning. Cambridge University Press (2017)
15. Elaasar, M., Badreddin, O.: Modeling meets programming: A comparative study in model driven engineering action languages. In: Margaria and Steffen [38], pp. 50–67, doi: 10.1007/978-3-319-47169-3_5
16. Elmqvist, H., Henningsson, T., Otter, M.: Systems modeling and programming in a unified environment based on Julia. In: Margaria and Steffen [38], pp. 198–217, doi: 10.1007/978-3-319-47169-3_15
17. ForMath project team: Papers and slides from the "formalisation of mathematics" (ForMath) project, available from http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/PapersAndSlides
18. Fritzson, P.: Principles of object-oriented modeling and simulation with Modelica 2.1. John Wiley & Sons (2010)
19. Halmos, P.: Naive Set Theory. Van Nostrand (1960), reprinted by Springer-Verlag, Undergraduate Texts in Mathematics, 1974
20. Haxthausen, A.E., Peleska, J.: On the feasibility of a unified modelling and programming paradigm. In: Margaria and Steffen [38], pp. 32–49, doi: 10.1007/978-3-319-47169-3_4
21. Igried, B., Setzer, A.: Programming with monadic CSP-style processes in dependent type theory. In: Proceedings of the 1st International Workshop on Type-Driven Development. pp. 28–38. TyDe 2016, ACM, New York, NY, USA (2016), doi: 10.1145/2976022.2976032, http://doi.acm.org/10.1145/2976022.2976032
22. Ionescu, C.: Vulnerability modelling and monadic dynamical systems. Ph.D. thesis, Freie Universität Berlin (2009)
23. Ionescu, C.: Vulnerability modelling with functional programming and dependent types. Mathematical Structures in Computer Science **26**(01), 114–128 (2016), doi: 10.1017/S0960129514000139

24. Ionescu, C., Jansson, P.: Domain-specific languages of mathematics: Presenting mathematical analysis using functional programming. In: Proceedings of the 4th and 5th International Workshop on Trends in Functional Programming in Education, TFPIE 2016, Sophia-Antipolis, France and University of Maryland College Park, USA, 2nd June 2015 and 7th June 2016. pp. 1–15 (2016), doi: 10.4204/EPTCS.230.1
25. Jansson, P., Einarsdóttir, S.H., Ionescu, C.: Examples and results from a bsc-level course on domain specific languages of mathematics. In: Proc. 7th Int. Workshop on Trends in Functional Programming in Education. EPTCS, Open Publishing Association (2018), in submission. Presented at TFPIE 2018.
26. Jansson, P., Ionescu, C.: Domain specific languages of mathematics: Lecture notes (2018), available from https://github.com/DSLsofMath/DSLsofMath
27. Jeffrey, A.: Dependently typed web client applications. In: Sagonas, K. (ed.) Practical Aspects of Declarative Languages (PADL). pp. 228–243. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), doi: 10.1007/978-3-642-45284-0_16
28. Kugler, H.: Unifying modelling and programming: A systems biology perspective. In: Margaria and Steffen [38], pp. 131–133, doi: 10.1007/978-3-319-47169-3_10
29. Lamport, L.: How to write a 21st century proof. Journal of Fixed Point Theory and Applications (November 2011), doi: 10.1007/s11784-012-0071-6, https://www.microsoft.com/en-us/research/publication/write-21st-century-proof/
30. Landin, P.J.: The next 700 programming languages. Communications of the ACM **9**(3), 157–166 (March 1966)
31. Larsen, P.G., Fitzgerald, J.S., Woodcock, J., Nilsson, R., Gamble, C., Foster, S.: Towards semantically integrated models and tools for cyber-physical systems design. In: Margaria and Steffen [38], pp. 171–186, doi: 10.1007/978-3-319-47169-3_13
32. Larsen, P.G., Lausdahl, K., Battle, N., Fitzgerald, J., Wolff, S., Sahara, S., Verhoef, M., Tran-Jørgensen, P.W.V., Oda, T.: VDM-10 Language Manual. Tech. Rep. TR-001, The Overture Initiative, www.overturetool.org (April 2013)
33. Lattmann, Z., Kecskés, T., Meijer, P., Karsai, G., Völgyesi, P., Lédeczi, Á.: Abstractions for modeling complex systems. In: Margaria and Steffen [38], pp. 68–79, doi: 10.1007/978-3-319-47169-3_6
34. Leavens, G.T., Naumann, D.A., Rajan, H., Aotani, T.: Specifying and verifying advanced control features. In: Margaria and Steffen [38], pp. 80–96, doi: 10.1007/978-3-319-47169-3_7
35. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7), 107–115 (2009), doi: 10.1145/1538788.1538814
36. Lethbridge, T.C., Abdelzad, V., Orabi, M.H., Orabi, A.H., Adesina, O.: Merging modeling and programming using Umple. In: Margaria and Steffen [38], pp. 187–197, doi: 10.1007/978-3-319-47169-3_14
37. Ludewig, J.: Models in software engineering – an introduction. Softw. Syst. Model **2**, 5–14 (2003), doi: 10.1007/s10270-003-0020-3
38. Margaria, T., Steffen, B. (eds.): Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II, Lecture Notes in Computer Science, vol. 9953 (2016), doi: 10.1007/978-3-319-47169-3
39. Marlow, S.: Parallel and concurrent programming in Haskell. In: Central European Functional Programming School: 4th Summer School, CEFP 2011, Budapest, Hungary, June 14–24, 2011, Revised Selected Papers. pp. 339–401. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), doi: 10.1007/978-3-642-32096-5_7

40. MBE Visual Glossary project: Model-based engineering visual glossary. See http://modelbasedengineering.com/glossary/ (2017)
41. Morgenstern, J., Licata, D.: Security-typed programming within dependently-typed programming. In: International Conference on Functional Programming. ACM (2010), doi: 10.1145/1863543.1863569
42. Naujokat, S., Neubauer, J., Margaria, T., Steffen, B.: Meta-level reuse for mastering domain specialization. In: Margaria and Steffen [38], pp. 218–237, doi: 10.1007/978-3-319-47169-3_16
43. Object Management Group (OMG): Unified modeling language. OMG Document Number formal/17-12-05 (https://www.omg.org/spec/UML/2.5.1/) (2017)
44. Oury, N., Swierstra, W.: The power of Pi. In: Proc. of ICFP 2008. pp. 39–50. ACM (2008), doi: 10.1145/1411204.1411213
45. Pierce, B.C.: Types and Programming Languages. MIT Press, 1st edn. (2002)
46. Prinz, A., Møller-Pedersen, B., Fischer, J.: Modelling and testing of real systems. In: Margaria and Steffen [38], pp. 119–130, doi: 10.1007/978-3-319-47169-3_9
47. Protzenko, J., Zinzindohoué, J.K., Rastogi, A., Ramananandro, T., Wang, P., Zanella-Béguelin, S., Delignat-Lavaud, A., Hriţcu, C., Bhargavan, K., Fournet, C., Swamy, N.: Verified low-level programming embedded in F*. Proc. ACM Program. Lang. **1**(ICFP), 17:1–17:29 (Aug 2017), doi: 10.1145/3110261, http://arxiv.org/abs/1703.00053
48. Rouquette, N.F.: Simplifying OMG mof-based metamodeling. In: Margaria and Steffen [38], pp. 97–118, doi: 10.1007/978-3-319-47169-3_8
49. Rybicki, F., Smyth, S., Motika, C., Schulz-Rosengarten, A., von Hanxleden, R.: Interactive model-based compilation continued - incremental hardware synthesis for SCCharts. In: Margaria and Steffen [38], pp. 150–170, doi: 10.1007/978-3-319-47169-3_12
50. Seidewitz, E.: On a unified view of modeling and programming position paper. In: Margaria and Steffen [38], pp. 27–31, doi: 10.1007/978-3-319-47169-3_3
51. Selic, B.: Programming $\subset$ modeling $\subset$ engineering. In: Margaria and Steffen [38], pp. 11–26, doi: 10.1007/978-3-319-47169-3_2
52. Swamy, N., Chen, J., Fournet, C., Strub, P.Y., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. In: Proc. of ICFP 2011. pp. 266–278 (2011), doi: 10.1145/2034773.2034811
53. Turner, R.: Computable Models. Springer-Verlag, 1 edn. (2009), doi: 10.1007/978-1-84882-052-4
54. Wells, C.: Communicating mathematics: Useful ideas from computer science. American Mathematical Monthly pp. 397–408 (1995), doi: 10.2307/2975030